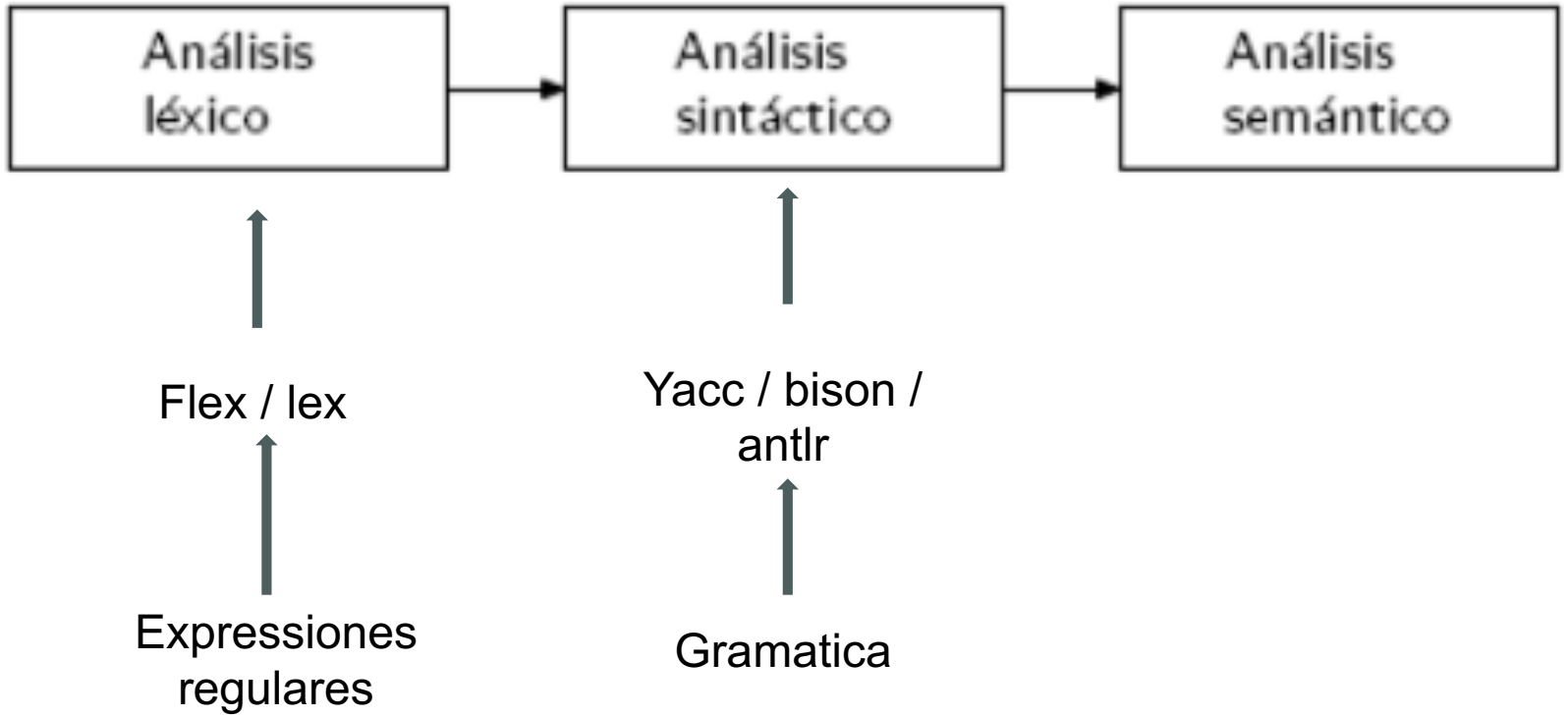
CS3025 Compiladores

Semana 2: Analisis Lexico

21 Agosto 2023

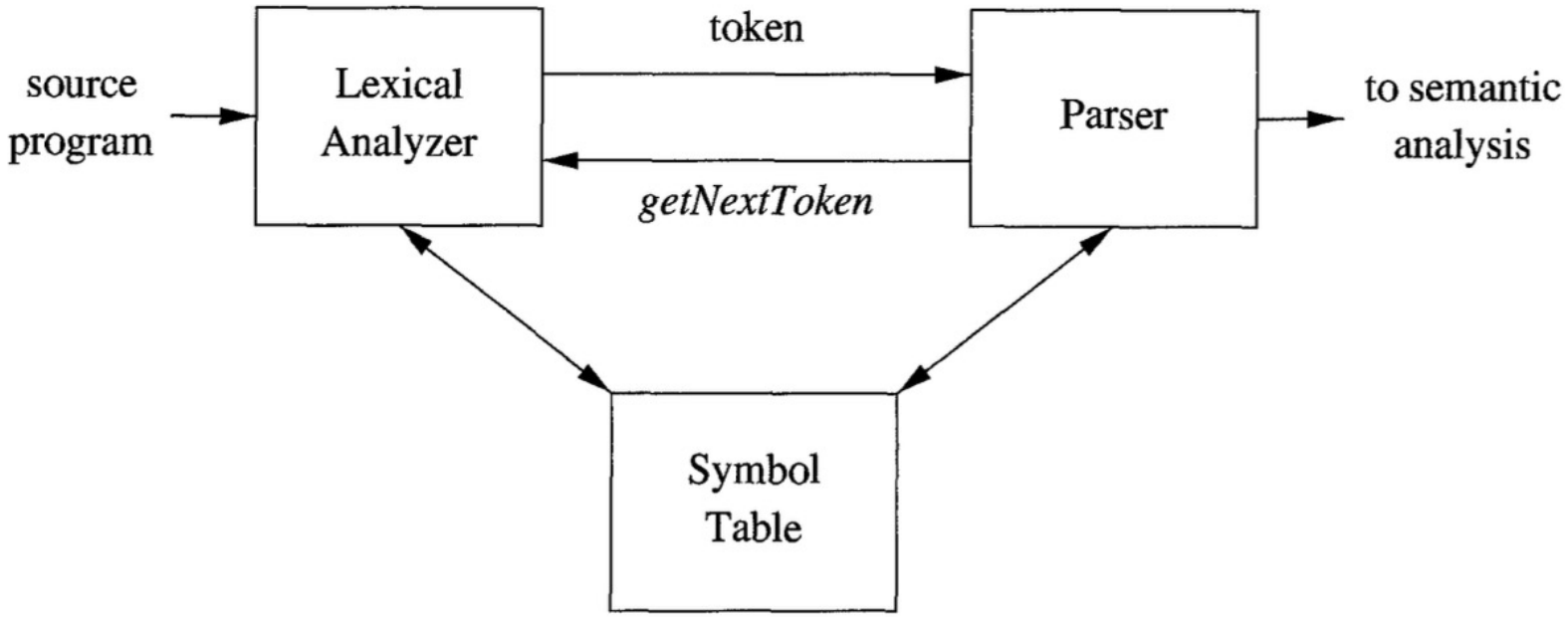
Igor Siveroni

Front-End de un compilador



Analisis Lexico

El objetivo del análisis léxico es el de leer una secuencia de caracteres (programa fuente) y producir como output una secuencia de tokens (que el Parser utilizara para realizar el análisis sintáctico)



Analisis Lexico: Tokens, Patrones y Lexemas

- **Token**: Representa un tipo de unidad lexica definida por la sintaxis del lenguaje.
  Consiste en un nombre y un valor opcional (atributo). El nombre del token es un símbolo abstracto que representa el tipo de unidad lexica e.g. secuencias de caracteres denotando identificadores o números.

- **Patrón / Pattern**: Descripción que indica la forma de los lexemas pertenecientes a un token (o unidad lexica).
  Patrones son especificados por Expresiones Regulares.

- **Lexema**: Secuencia de caracteres del programa fuente que corresponde (match) al patrón de un token y es identificado por el análisis léxico como instancia de ese token.

Expresiones Regulares: Formalismo para especificar patrones

Una expresión regular r define el lenguaje L(r). Las reglas que definen las expresiones regulares validas sobre un alfabeto Σ son:

- ϵ es una expresión regular, y L(ϵ) = { ϵ }.
- Si a es un símbolo en Σ, entonces a es una exp. Regular, y L(a) = { a }.

Dados lenguajes regulares r y s:

- `r | s` es un lenguaje regular que denota el lenguaje L(r) U L(s).
- `r s` es un lenguaje regular que denota el lenguaje L(r) L(s).
- r* es un lenguaje regular que denota L(r)*
- (r) es un lenguaje regular que denota L(r) --- paréntesis!

El operador * tiene la más alta precedencia, seguido por concatenación. | tiene la más baja precedencia. Todos asocian a la izquierda.

Expresiones Regulares: usar definiciones regulares

La expresión regular *id* usa las definiciones regulares *letter_* y *digit*.

$$
\begin{array}{rcl}
\mathit{letter\_} & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{id} & \rightarrow & \mathit{letter\_} \; (\; \mathit{letter\_} \mid \mathit{digit} \;)^*
\end{array}
$$

De manera similar *number* usa definiciones regulares:

$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{digits} & \rightarrow & \mathit{digit} \; \mathit{digit}^* \\
\mathit{optionalFraction} & \rightarrow & \texttt{.} \; \mathit{digits} \mid \epsilon \\
\mathit{optionalExponent} & \rightarrow & (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; \mathit{digits} \;) \mid \epsilon \\
\mathit{number} & \rightarrow & \mathit{digits} \; \mathit{optionalFraction} \; \mathit{optionalExponent}
\end{array}
$$

Scanner / Rastreador: El codigo

El Scanner / Rastreador es un programa que examina una cadena (secuencia de caracteres) y encuentra un prefijo que es el lexema que corresponde (match) a uno de los patrones de las unidades léxicas.

Es un proceso secuencial, de izquierda a derecha, que inspecciona un carácter a la vez. Este proceso maneja (al menos de manera abstracta), 2 punteros: el puntero actual y el puntero de inicio de lexema (donde se empezó a leer el lexema actual)

Usaremos las siguientes funciones auxiliares:

- c = nextChar(): retorna el siguiente carácter de la secuencia y avanza el puntero una posición.
- rollBack(): (retract) Retrocede el puntero una posición. “devuelve el carácter leído”.
- startLexema(): Marca el inicio del siguiente lexema.
- getLexema(): Retorna el lexema actual (marcado por el puntero de inicio de lexema y el puntero actual)

Scanner: Reconociendo patrones

Con la ayuda de las funciones auxiliares podemos reconocer una serie de patrones (expresiones regulares) comunes. Algunos ejemplos:

- Reconocer un carácter e.g. el operador

```
c =nextChar();
if (c == '+') return new Token(Token::PLUS, c)
```

- Consumir espacios en blanco. ws ::= (blank | tab | newline)+

```
c =nextChar();
while (c == ' ' || c == '\t' || c == '\n')  c = nextChar();
```

- Numeros. num ::= digit digit+

```
c =nextChar();
if (isdigit(c)) {
   c = nextChar(); while (isdigit(c)) c = nextChar();
   rollBack();
   token = new Token(Token::NUM, getLexema());
```

Scanner: Reconociendo patrones

Y escribir el scanner para el mini lenguaje de expresiones aritméticas de la siguiente manera (`scanner_aexp.cpp`):

```
Token* Scanner::nextToken() {
  Token* token;
  char c;
  // consume whitespaces
  c = nextChar();
  while (c == ' ') c = nextChar();
  if (c == '\0') return new Token(Token::END);
  startLexema();
  if (isdigit(c)) {
    c = nextChar();
    while (isdigit(c)) c = nextChar();
    rollBack();
    token = new Token(Token::NUM, getLexema());
  } else if (strchr("+-*/^()", c)) {
    switch(c) {
    case '(': token = new Token(Token::LPAREN); break;
    case ')': token = new Token(Token::RPAREN); break;
    case '+': token = new Token(Token::PLUS,c); break;
    case '-': token = new Token(Token::MINUS,c); break;
    case '*': token = new Token(Token::MULT,c); break;
    case '/': token = new Token(Token::DIV,c); break;
    case '^': token = new Token(Token::POW,c); break;
    default: cout << "No deberia llegar aca" << endl;
    }
  } else {
    token = new Token(Token::ERR, c);
  }
  return token;
}
```

Programacion de scanner: Automatizando el proceso

Veremos que una de las mejores maneras de definir procesos para la identificación de expresiones regulares es la construccion de diagramas de transición (automata).

Trabajemos con el siguiente ejemplo, una gramática de branching statements:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
 & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
 & | & \epsilon \\
\textit{expr} & \rightarrow & \textit{term} \textbf{ relop } \textit{term} \\
 & | & \textit{term} \\
\textit{term} & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

Veremos en las siguientes semanas lo que significa esta gramática. Por ahora, nos interesa la definición de los terminales usados por la gramática (unidades léxicas / tokens)

Ejemplo: branching statements

Los patrones de los tokens del lenguaje son:

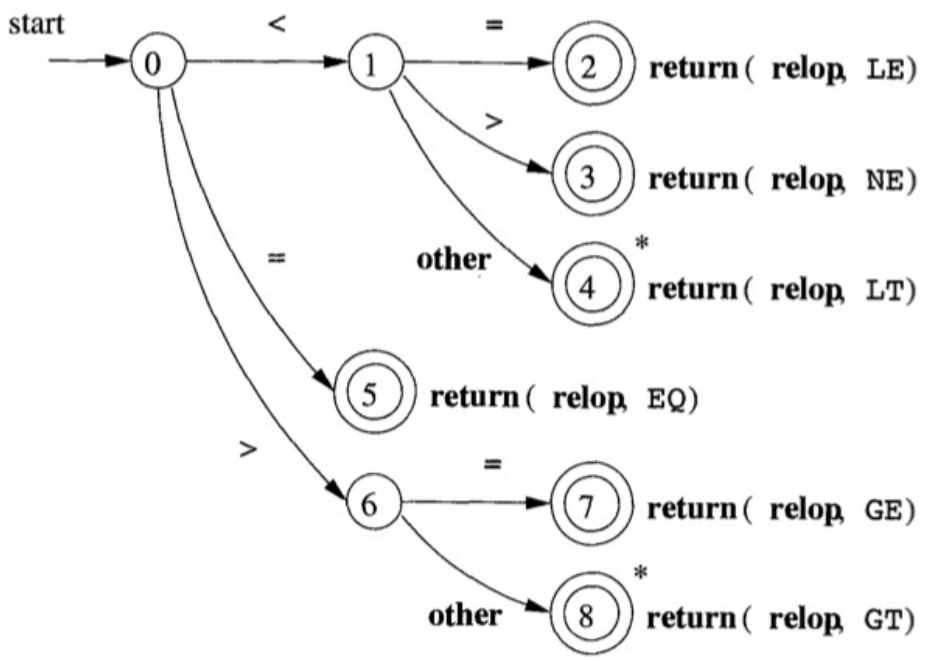
$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & \texttt{[0-9]} \\
\mathit{digits} & \rightarrow & \mathit{digit}^{+} \\
\mathit{number} & \rightarrow & \mathit{digits}\ (\texttt{.}\ \mathit{digits})?\ (\ \texttt{E}\ \texttt{[+-]}?\ \mathit{digits}\ )? \\
\mathit{letter} & \rightarrow & \texttt{[A-Za-z]} \\
\mathit{id} & \rightarrow & \mathit{letter}\ (\ \mathit{letter} \mid \mathit{digit}\ )^{*} \\
\mathit{if} & \rightarrow & \texttt{if} \\
\mathit{then} & \rightarrow & \texttt{then} \\
\mathit{else} & \rightarrow & \texttt{else} \\
\mathit{relop} & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

Además, asignamos al scanner la labor de remover los espacios en blanco (y similares) al crear un token específico para esto:

$$ws \rightarrow (\ \mathbf{blank} \mid \mathbf{tab} \mid \mathbf{newline}\ )^{+}$$

Ejemplo: branching statements

El siguiente paso es el de convertir los patrones a diagramas de estado.
Empecemos con el diagrama que representa la aceptación de operadores relacionales (relOp)



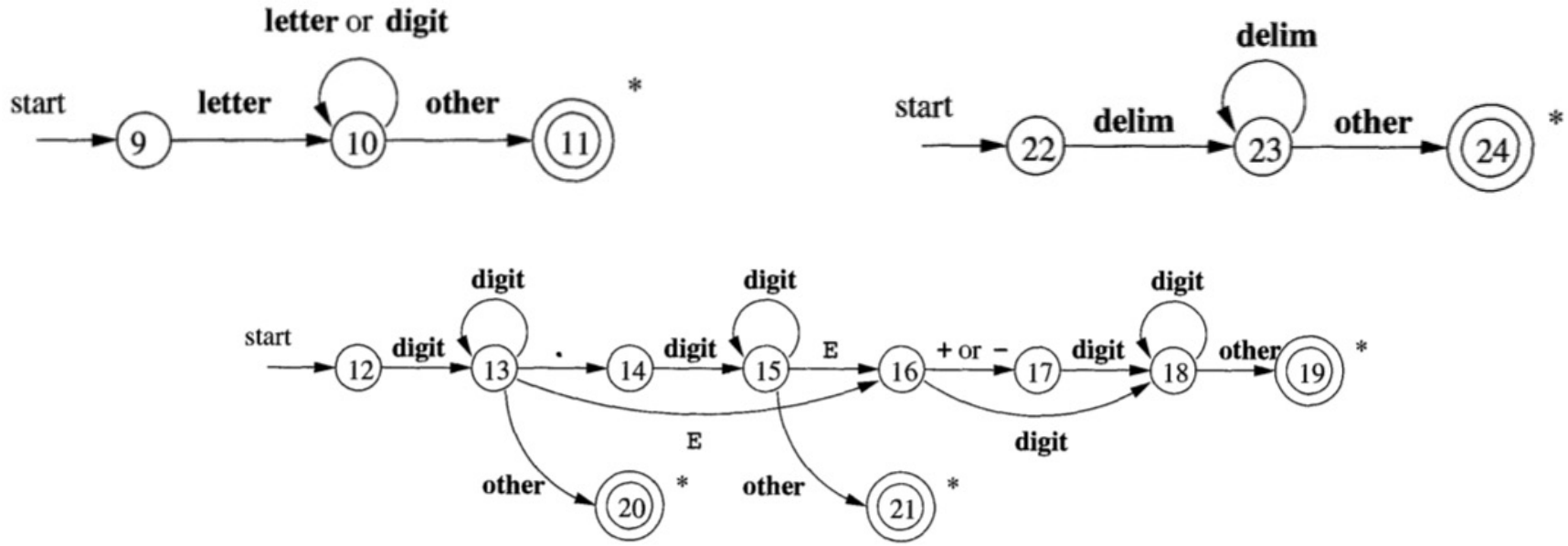
Al entrar al estado inicial (0) se activa el seguimiento del lexema: startLexema().
El scanner siempre busca el lexema mas largo que cumpla con el patron. Nuestro diagrama logra esto al implementar un tipo de lookahead e.g. estado 3-4, donde se lee el carácter pero se ‘devuelve’ si no hay match. Esto se denota con el Asterisco *.

Ejemplo: branching statements – scanning relOp

El siguiente código implementa el diagrama anterior para verificar si la cadena de imput tiene como prefijo un relOp (archivo: `scanner_relop.cpp`):

```
bool Scanner::checkRelOp() {
  char c;
  state = 0;
  startLexema();
  while(1) {
    switch(state) {
    case 0:
      c = nextChar();
      if (c == '<') state =1;
      else if (c == '=') state = 5;
      else if (c == '>') state = 6;
      else return false;
      break;
    case 1:
      c = nextChar();
      if (c == '=') state =2;
      else if (c == '>') state = 3;
      else  state = 4;
      break;
    case 2: return true;
    case 3: return true;
    case 4: rollBack(); return true;
    case 5: return true;
    case 6:
      c = nextChar();
      if (c == '=') state =7;
      else  state = 8;
      break;
    case 7: return true;
    case 8: rollBack(); return true;
    }
  }
}
```

Ejemplo: branching statements – el resto de diagramas



El scanner trabaja con un diagrama de transición que integra los diagramas de transición de todos los patrones.

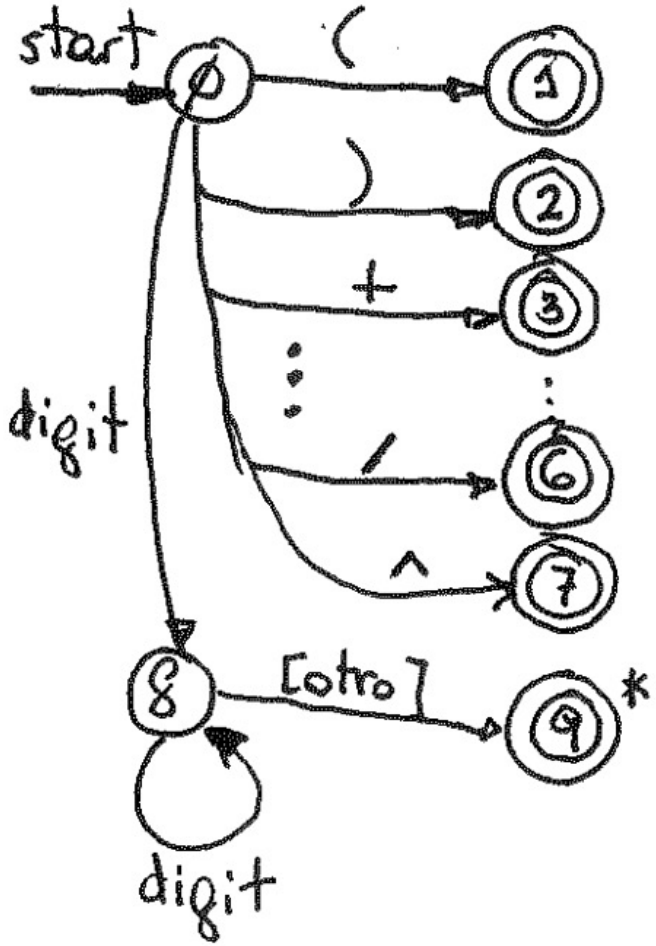
Ejemplo: branching statements – Unificar los diagramas

Ejercicio:
Unificar los diagramas e implementar un scanner basado en el diagrama de transición final

…

Como muestra y ejemplo, re-implementemos el scanner usado para analizar expresiones aritméticas

Expresiones aritmeticas: Diagrama de Transicion



Scanner: Expresiones aritmeticas basado en diagramas de transicion

```
Token* Scanner::nextToken() {
  Token* token;
  char c;
  state = 0;
  startLexema();
  while (1) {
    switch (state) {
    case 0: c = nextChar();
      if (c == ' ') {
         incrStartLexema(); state = 0; }
      else if (c == '\0')
         return new Token(Token::END);
      else if (c == '(') state = 1;
      else if (c == ')') state = 2;
      else if (c == '+') state = 3;
      else if (c == '-') state = 4;
      else if (c == '*') state = 5;
      else if (c == '/') state = 6;
      else if (c == '^') state = 7;
      else if (isdigit(c)) { state = 8; }
      else return new Token(Token::ERR, c);
      break;
```

```
    case 1: return new Token(Token::LPAREN);
     case 2: return new Token(Token::RPAREN);
     case 3: return new Token(Token::PLUS,c);
     case 4: return new Token(Token::MINUS,c);
     case 5: return new Token(Token::MULT,c);
     case 6: return new Token(Token::DIV,c);
     case 7: return new Token(Token::POW,c);
     case 8: c = nextChar();
       if (isdigit(c)) state = 8;
       else state = 9;
       break;
     case 9: rollBack();
       return new Token(Token::NUM, getLexema());
       }
    }
}
```

Acrhivo: **scanner_aexp_automaton.cpp**

Ejemplo: branching statements – Unificar los diagramas

La concatenación de diagramas de transición generalmente se realiza introduciendo ϵ (cadena vacía) y otras fuentes de no-determinismo.

El automaton final es generalmente no-determinista (NFA)

Es complicado generar código directamente de un NFA

Existen algoritmos que transforman NFAs a DFAs.

Eso no será parte de este curso! Pero a continuación, un ejemplo.

Ejemplo: Expresiones regulares a NFAs y DFAs (Appel cap. 2)

Consideremos las siguientes expresiones regulares que definen los tokens de un lenguaje:

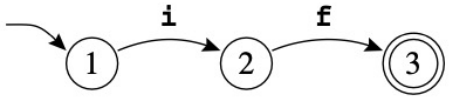
`if`	IF
`[a-z][a-z0-9]*`	ID
`[0-9]+`	NUM
`([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)`	REAL
`("--"[a-z]*"\n")|(" "|"\n"|"\t")+`	*no token, just white space*
`.`	*error*

Nótese que las reglas son algo ambiguas. Por ejemplo, `if8` es un identificador o 2 tokens `if` y `8`? La cadena `if 89` empieza con un identificador o una palabra reservada? Analizadores léxicos utilizan 2 criterios para eliminar estas ambigüedades:

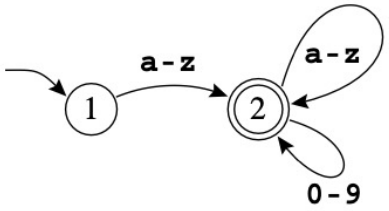
- Cadena mas larga: Elije el token que hace match con la subcadena mas larga (`if8` es ID)
- Prioridad de reglas: Si persiste la ambigüedad, se elije a la regla con mayor prioridad, es decir, la que aparece primero en la lista (`if` hace match como palabra reservada).

Ejemplo: Expresiones regulares a NFAs y DFAs (Appel cap. 2)

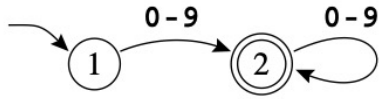
Los siguientes autómatas finitos corresponden a los tokens (expresiones regulares) definidos en la pagina anterior:



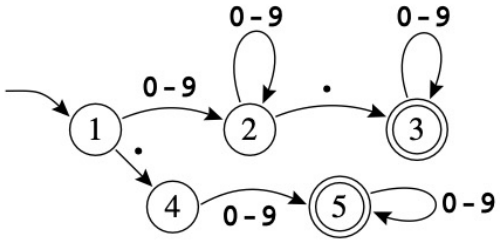
IF



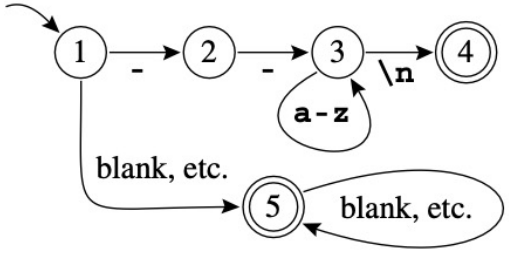
ID



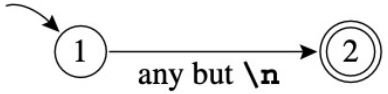
NUM



REAL



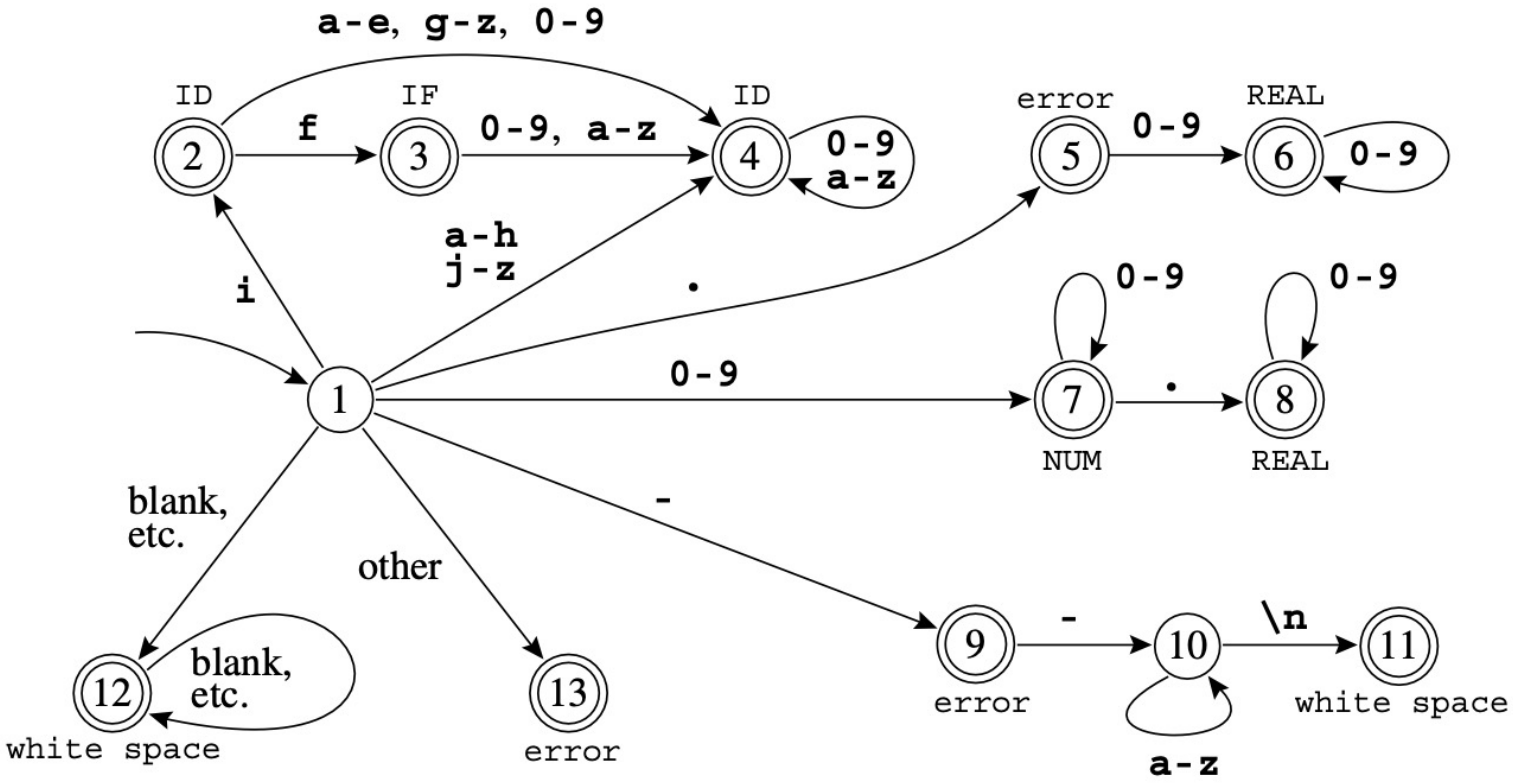
white space



error

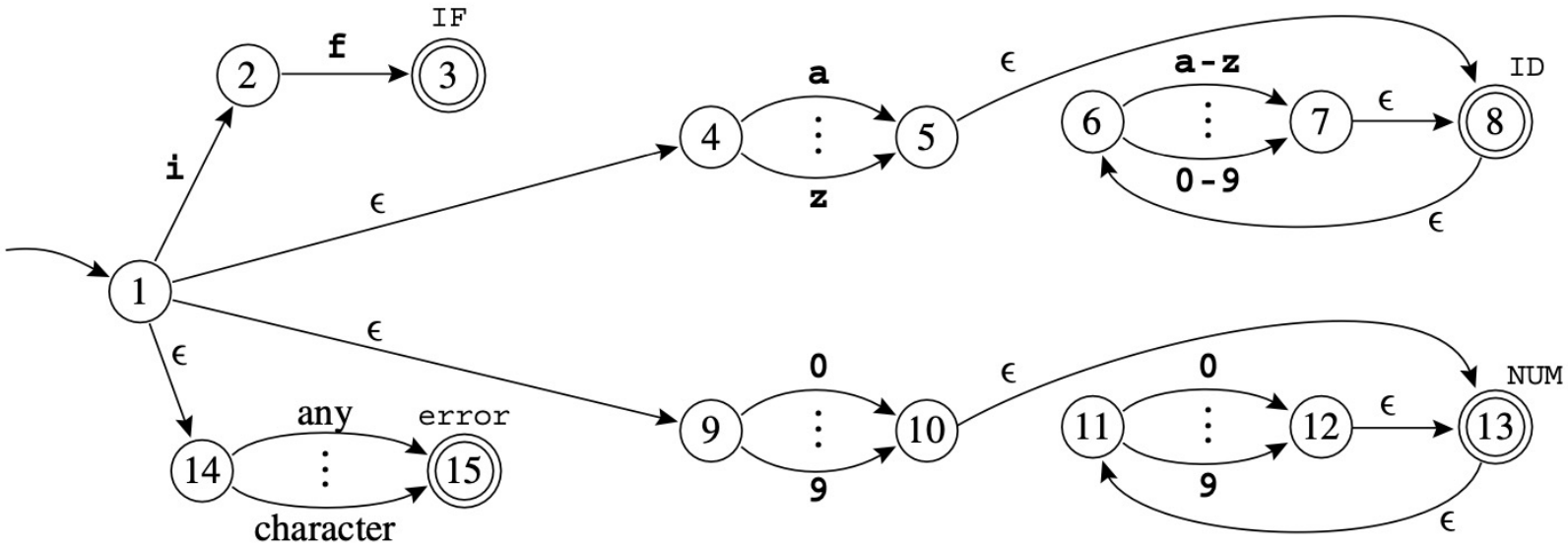
Ejemplo: Expresiones regulares a NFAs y DFAs (Appel cap. 2)

Si combinamos los diagramas de la pagina anterior, obtenemos el siguiente automata finito (determinístico):



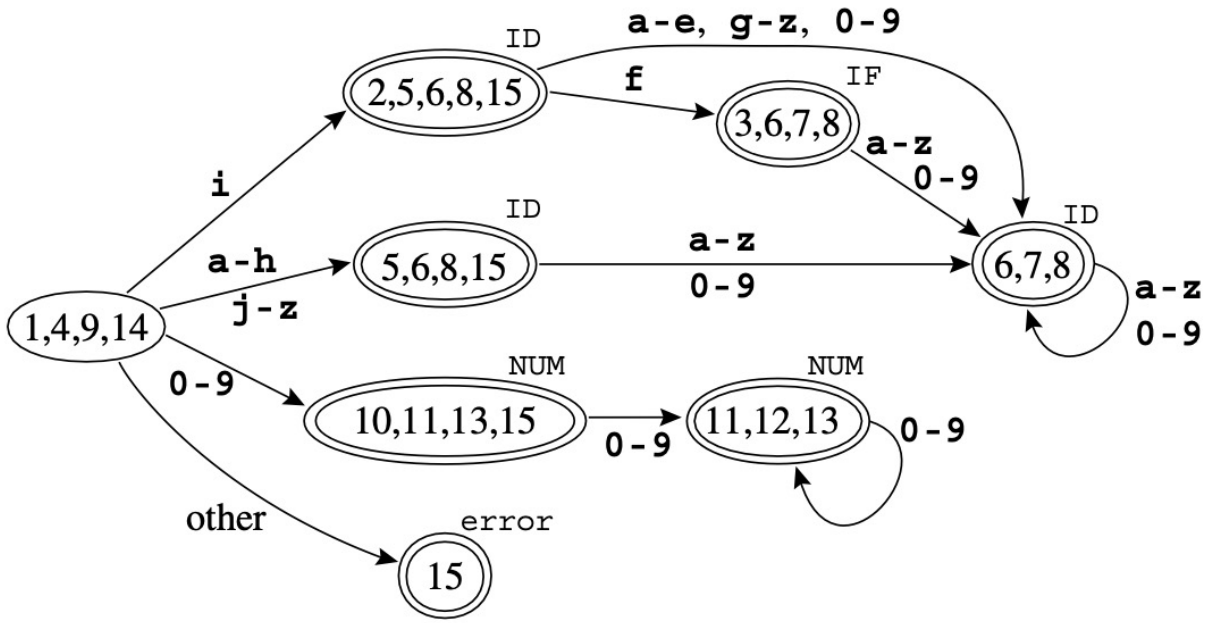
Ejemplo: Expresiones regulares a NFAs y DFAs (Appel cap. 2)

Si combinamos los automatas originales utilizando técnicas de creación de automatas desde expresiones regulares, obtenemos lo siguiente:



Ejemplo: Expresiones regulares a NFAs y DFAs (Appel cap. 2)

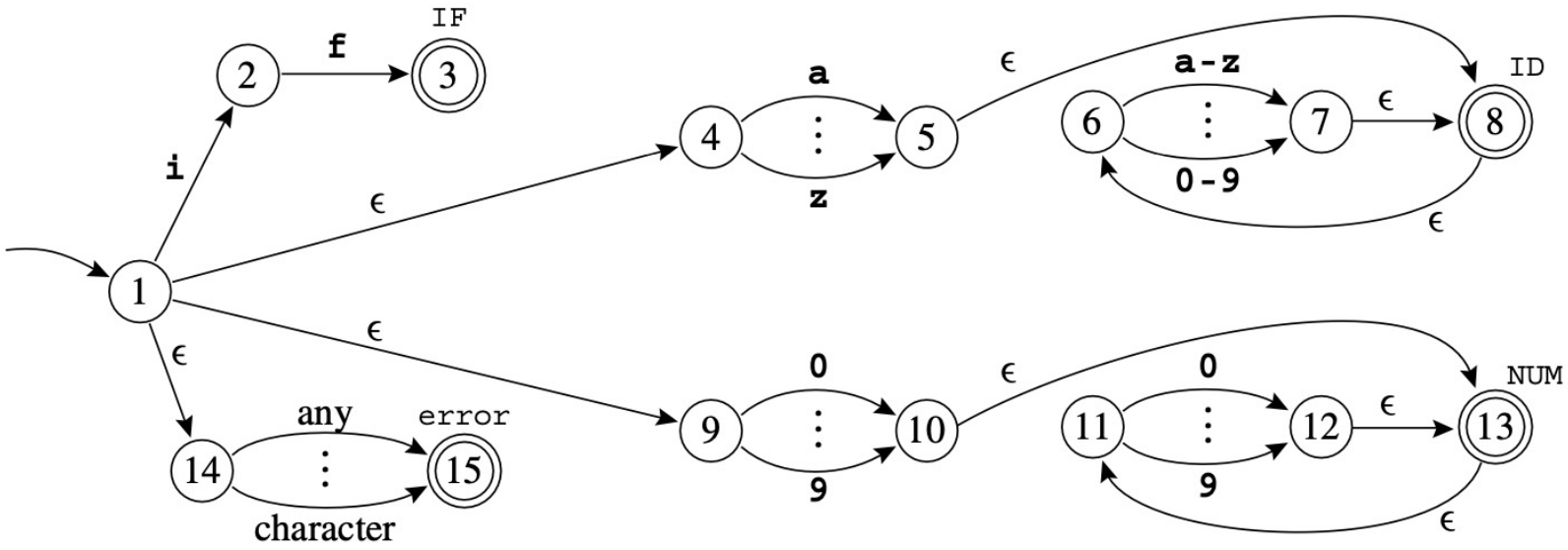
La transformación anterior genero un automata no determinisco. Utilizando algoritmos conocidos que remueven no-determinismo obtenemos el siguiente automata



OK! Podemos trabajar con este automata. Sin embargo, el automata es subóptimo (e.g. tamaño). Existen algoritmos para reducción / optimizacion de automatas … pero hasta aca llegamos

Ejemplo: Expresiones regulares a NFAs y DFAs (Appel cap. 2)

Si combinamos los automatas originales utilizando técnicas de creación de automatas desde expresiones regulares, obtenemos lo siguiente:



Analisis Lexico: Observaciones

Si bien es correcto y eficiente implementar *scanners* utilizando autómatas como referencia, llega un punto - si consideramos lenguajes con especificaciones léxicas extensas - que el proceso se hace tedioso si se hace a mano.

Hemos visto que este proceso, de expresiones regulares a autómatas, puede ser automatizado. Esta es la razón de ser des Generadores de Analizadores Lexicos (Lexical Analyzer Generators)

Mejor: Usar herramientas que generen scanners desde expresiones regulares
Usaremos FLEX!!

Nota: Palabras Reservadas

- Las palabras reservadas generalmente son instancias de la unidad lexica ID de nuestro lenguaje.g. if, then, else cumplen las reglas del patron ID.
- No tiene sentido escribir código que reconozca cada una de las palabras reservadas.
- Es mejor tener un registro de las palabras reservadas (hash table) y,
- Cada vez que encontramos un ID, verificar si el lexema se encuentra en la tabla. Si es asi, creamos el token correspondiente a la palabra reservada, de lo contrario, creamos el token ID.

Analisis Lexico: Lecturas

Andrew Appel.
Modern Compiler Implementation in Java.
Capitulo 2 (Subido a Canvas)

Alfred Aho, Ravi Sathi and Jeffrey Ullman.
Compilers: Principles, Techniques and Tools.
Capitulo 3.